

PONTIFÍCIA UNIVERSIDADE CATÓLICA DO RIO GRANDE
DO SUL
BACHARELADO EM ENGENHARIA DE SOFTWARE

Algoritmos e Estrutura de Dados II:
Trabalho 2

GABRIEL FERREIRA E HENRIQUE DA SILVA JUCHEM
PORTO ALEGRE
2023

Problema sendo resolvido:

O problema apresentado no trabalho se deu pela criação de um mapa dos mares Egeu, Adriático e Mediterraneo onde serão realizadas viagens de transporte de mercadorias. Após a saída do navio serão realizadas algumas paradas onde haverão 9 locais específicos onde eles terão que parar, seguindo sempre a ordem crescente de 1 a 9, assim após essas paradas o navio deverá voltar ao ponto de partida inicial. Em toda a região haverá um mapa onde a cada uso pode mudar de locais de partida e pontos de parada, assim nenhuma navegação será semelhante a outra no sistema, também haverá alterações nos mapas podendo haver conflitos com pontos de parada e o ponto de partida.

Como o problema foi modelado:

O problema foi modelado tendo em vista uma forma de percorrer um grafo não direcionado tendo como base o algoritmo de Dijkstra. Foi necessário criar um objeto para cada vértice, guardando-os em uma matriz onde para navegar entre eles foi utilizado um sistema que utilizava das linhas e colunas do texto como coordenadas.

Como é o processo de solução, apresentando exemplos e algoritmos:

Para a solução do trabalho foi utilizado o algoritmo de Dijkstra vista em aula:

```

public class Dijkstra {
    public int[] antecessor;
    public int[] distancia;
    public boolean[] percorrido;
    private Grafo grafo;

    public Dijkstra(Grafo g, int origem) {
        this.grafo = g;
        antecessor = new int[grafo.getNumeroVertices()];
        distancia = new int[grafo.getNumeroVertices()];
        percorrido = new boolean[grafo.getNumeroVertices()];
        for (int i = 0; i < grafo.getNumeroVertices(); i++) {
            antecessor[i] = -1;
            distancia[i] = Integer.MAX_VALUE;
            percorrido[i] = false;
        }

        FilaPrioridadeMinima filaMin = new FilaPrioridadeMinima();
        filaMin.enfileirar(origem, distancia[0]);
        distancia[origem] = 0;

        while (!filaMin.estaVazia()) {
            int vertice = filaMin.desenfileirar();
            percorrido[vertice] = true;
            for (int aresta : g.adjacentes(vertice)) {
                int destino = aresta;
                int distanciaDestino = distancia[vertice] + 1;
                if (distanciaDestino < distancia[destino]) {
                    antecessor[destino] = vertice;
                    distancia[destino] = distanciaDestino;
                    if (!filaMin.existe(destino))
                        filaMin.enfileirar(destino, distanciaDestino);
                    else
                        filaMin.atualizarDistancia(destino, distanciaDestino);
                }
            }
        }
    }

    public int getDistancia(int destino) {
        int distancia = this.distancia[destino];
        if (distancia == Integer.MAX_VALUE) {
            throw new RuntimeException(message: "Não existe caminho entre os vértices");
        }
        return distancia;
    }
}

```

e a classe grafo presente no trabalho extra:

```

import java.util.ArrayList;

public class Grafo {
    private ArrayList<Integer>[] listaAdjacencia;
    private int numeroVertices;
    private int numeroArestas;

    public Grafo(int numeroVertices) {
        this.numeroVertices = numeroVertices;
        this.numeroArestas = 0;
        listaAdjacencia = new ArrayList[numeroVertices];
        for (int i = 0; i < numeroVertices; i++) {
            listaAdjacencia[i] = new ArrayList<>();
        }
    }

    public void adicionarAresta(int v, int w) {
        if (!existeAresta(v, w)) {
            listaAdjacencia[v].add(w);
            listaAdjacencia[w].add(v);
            numeroArestas++;
        }
    }

    public boolean existeAresta(int v, int w) {
        boolean vw = listaAdjacencia[v].indexOf(w) >= 0;
        boolean wv = listaAdjacencia[w].indexOf(v) >= 0;
        return vw || wv;
    }

    public void removerAresta(int v, int w) {
        int vw = listaAdjacencia[v].indexOf(w);
        listaAdjacencia[v].remove(vw);
        int wv = listaAdjacencia[w].indexOf(v);
        listaAdjacencia[w].remove(wv);
    }

    public ArrayList<Integer> adjacentes(int v) {
        return listaAdjacencia[v];
    }

    public String toDot() {
        String resultado = "graph G { " + System.lineSeparator();
        for (int i = 0; i < numeroVertices; i++) {
            resultado = resultado + "\t" + i + ";" + System.lineSeparator();
        }
        for (int i = 0; i < numeroVertices; i++) {
            for (int j = 0; j < listaAdjacencia[i].size(); j++) {
                resultado += "\t" + i + "--" + listaAdjacencia[i].get(j) + ";" + System.lineSeparator();
            }
        }
        resultado += "}";
        return resultado;
    }

    public int getNumeroVertices() {
        return this.numeroVertices;
    }
}

```

Primeiramente é feita a leitura do mapa.

```

try {
    Scanner sc = new Scanner(new File(pathname:"exemplos/mapa_500_1000.txt"));
    String linhaUm = sc.nextLine();
    String[] arrayLinhaUm = (linhaUm.split(regex:" "));

    this.numeroDeColunas = Integer.parseInt(arrayLinhaUm[1]);
    this.numeroDeLinhas = Integer.parseInt(arrayLinhaUm[0]);

    matrizMapa = new char[numeroDeLinhas][numeroDeColunas];
    vertices = new Vertice[numeroDeLinhas * numeroDeColunas];
    grafo = new Grafo(numeroDeLinhas * numeroDeColunas);

    int contador = 0;

```

É salvo o número de colunas e linhas para montar o tamanho da matriz do mapa, além dos vértices e do grafo. É inicializado um contador que vai servir como índice para cada vértice.

```

while (sc.hasNextLine()) {
    for (int i = 0; i < matrizMapa.length; i++) {
        String linha = sc.nextLine();
        for (int j = 0; j < matrizMapa[i].length; j++) {
            matrizMapa[i][j] = linha.charAt(j);
            Vertice v = new Vertice(contador, i, j, matrizMapa[i][j]);
            this.vertices[contador] = v;
            contador++;
            try {
                int vertice = Integer.parseInt(String.valueOf(matrizMapa[i][j]));
                this.pontos1a9[vertice - 1] = v;
            } catch (NumberFormatException nfe) {
                continue;
            }
        }
    }
} catch (FileNotFoundException fnfe) {
    System.out.println(x:"Arquivo não encontrado!");
} catch (Exception e) {
    System.out.println(e);
}

```

Aqui são iniciados os vértices em suas respectivas coordenadas e também o vetor para guardar os pontos e seu lugar relativo no mapa.

Classe vértice:

```

public class Vertice {
    private int indice;
    private int linha;
    private int coluna;
    private char caracter;

    public Vertice(int indice, int linha, int coluna, char caracter) {
        this.indice = indice;
        this.linha = linha;
        this.coluna = coluna;
        this.caracter = caracter;
    }
}

```

Após isso o grafo é iniciado e é feita a ligação através do método abaixo:

```

public void ligar(Grafo grafo, int numeroDeLinhas, int numeroDeColunas) {
    for (int i = 0; i < numeroDeLinhas; i++) {
        for (int j = 0; j < numeroDeColunas; j++) {
            if (matrizMapa[i][j] == '*') {
                continue;
            }
            if (i > 0 && matrizMapa[i - 1][j] != '*') {
                grafo.adicionarAresta(i * numeroDeColunas + j, (i - 1) * numeroDeColunas + j);
            }
            if (i < numeroDeLinhas - 1 && matrizMapa[i + 1][j] != '*') {
                grafo.adicionarAresta(i * numeroDeColunas + j, (i + 1) * numeroDeColunas + j);
            }
            if (j > 0 && matrizMapa[i][j - 1] != '*') {
                grafo.adicionarAresta(i * numeroDeColunas + j, i * numeroDeColunas + j - 1);
            }
            if (j < numeroDeColunas - 1 && matrizMapa[i][j + 1] != '*') {
                grafo.adicionarAresta(i * numeroDeColunas + j, i * numeroDeColunas + j + 1);
            }
        }
    }
}

```

Esse método percorre o mapa e verifica nas quatro direções se existe alguma conexão de vértices, se o caracter do vértice for diferente de "*" ele adicionará a conexão como aresta para o grafo.

```

for (int i = 0; i < pontos1a9.length; i++) {
    Dijkstra inicio = new Dijkstra(grafo, pontos1a9[ondeEstou].getIndice());
    try {
        int d = inicio.getDistancia(pontos1a9[ondeVou].getIndice());
        ondeEstou = ondeVou;
        maisLonginquo = ondeEstou;
        novasDistancias[ondeVou] = d;
        ondeVou++;
    } catch (Exception ofb) {
        ondeVou++;
    }
}
}

```

Nessa parte do código o algoritmo Dijkstra é utilizado e é chamado o método `getDistancia` da mesma classe, que faz uma comparação para ver se há caminho entre vértices, caso haja é retornado a distancia mínima.

```

public int getDistancia(int destino) {
    int distancia = this.distancia[destino];
    if (distancia == Integer.MAX_VALUE) {
        throw new RuntimeException(message:"Não existe caminho entre os vértices");
    }
    return distancia;
}

```

O loop `for` percorre os elementos do array `pontos1a9`. Dentro do loop, uma instância da classe `Dijkstra` é criada com o nome de `inicio`, se não houver um caminho entre os pontos, esse método lançará uma exceção. Se a exceção não for lançada, a distância é armazenada no array `novasDistancias` na posição `ondeVou`. As variáveis `ondeEstou`, `maisLonginquo` e `ondeVou` são atualizadas, `ondeEstou` recebe o valor de `ondeVou`, `maisLonginquo` recebe o valor de `ondeEstou`, e `ondeVou` é incrementada. Se ocorrer uma exceção ao tentar obter a distância, o código incrementa `ondeVou` e continua para a próxima iteração do loop.

```

Dijkstra fim = new Dijkstra(grafo, pontos1a9[maisLonginquo].getIndice());
int d = fim.getDistancia(pontos1a9[0].getIndice());
novasDistancias[0] = d;

```

É calculado a distância mínima entre o ponto mais distante dos pontos de referência e o ponto de partida usando o algoritmo de Dijkstra.

Resultados dos casos de teste:

Mapa 15 linhas x 80 colunas

```
PROBLEMS 3 OUTPUT DEBUG CONSOLE TERMINAL COMMENTS

' '-cp' 'D:\Programming\GitHub\trab2Alest2\bin' 'App'
Não é possível chegar ao porto 2
A distancia do porto 3 é: 14
A distancia do porto 4 é: 3
A distancia do porto 5 é: 50
A distancia do porto 6 é: 57
A distancia do porto 7 é: 49
A distancia do porto 8 é: 40
A distancia do porto 9 é: 42
A distancia do ponto 0 é: 49
Combustível total gasto: 304
Tempo de execução: 59 ms
PS D:\Programming\GitHub\trab2Alest2>
```

Mapa 30 linhas x 80 colunas

```
PROBLEMS 3 OUTPUT DEBUG CONSOLE TERMINAL COMMENTS

xe' '-XX:+ShowCodeDetailsInExceptionMessages' '-cp' 'D:\
A distancia do porto 2 é: 82
A distancia do porto 3 é: 66
Não é possível chegar ao porto 4
Não é possível chegar ao porto 5
A distancia do porto 6 é: 38
A distancia do porto 7 é: 119
Não é possível chegar ao porto 8
A distancia do porto 9 é: 114
A distancia do ponto 0 é: 13
Combustível total gasto: 432
Tempo de execução: 111 ms
PS D:\Programming\GitHub\trab2Alest2>
```

Mapa 50 linhas x 100 colunas


```
PROBLEMS 3 OUTPUT DEBUG CONSOLE TERMINAL COMMENTS

xe' '-XX:+ShowCodeDetailsInExceptionMessages' '-cp' 'D:\Pr
A distancia do porto 2 é: 26
A distancia do porto 3 é: 74
Não é possível chegar ao porto 4
A distancia do porto 5 é: 53
A distancia do porto 6 é: 75
Não é possível chegar ao porto 7
A distancia do porto 8 é: 85
A distancia do porto 9 é: 108
A distancia do ponto 0 é: 63
Combustível total gasto: 484
Tempo de execução: 80 ms
PS D:\Programming\GitHub\trab2Alest2>
```

Mapa 60 linhas x 500 colunas

```
PROBLEMS 3 OUTPUT DEBUG CONSOLE TERMINAL COMMENTS

xe' '-XX:+ShowCodeDetailsInExceptionMessages'
A distancia do porto 2 é: 238
A distancia do porto 3 é: 58
A distancia do porto 4 é: 275
A distancia do porto 5 é: 110
A distancia do porto 6 é: 88
A distancia do porto 7 é: 39
A distancia do porto 8 é: 51
Não é possível chegar ao porto 9
A distancia do ponto 0 é: 127
Combustível total gasto: 986
Tempo de execução: 206 ms
PS D:\Programming\GitHub\trab2Alest2>
```

Mapa 500 linhas x 1000 colunas

```
PROBLEMS 3 OUTPUT DEBUG CONSOLE TERMINAL COMMENTS  
xe' '-XX:+ShowCodeDetailsInExceptionMessages' '-cp' 'D:\P  
A distancia do porto 2 é: 178  
A distancia do porto 3 é: 404  
A distancia do porto 4 é: 1017  
A distancia do porto 5 é: 390  
A distancia do porto 6 é: 389  
A distancia do porto 7 é: 174  
A distancia do porto 8 é: 282  
A distancia do porto 9 é: 294  
A distancia do ponto 0 é: 298  
Combustível total gasto: 3426  
Tempo de execução: 6469 ms  
PS D:\Programming\GitHub\trab2Alest2>
```

Conclusões:

Foi concluído que o algoritmo de Dijkstra era com certeza o mais apropriado e eficiente para realizar o trabalho. Também foi possível notar a simplicidade em atribuir algum tipo de identificação global, como foi obtida pelo índice de vértices. Em suma, o trabalho serviu para ampliar os conhecimentos sobre grafos e tipos de encaminhamentos úteis e compatíveis para serem utilizados. Além de mostrar a eficácia da estrutura de dados para problemas envolvendo algum tipo de conexão em geral.